第 7 章　查 找

7.1 概述

查找和排序是软件设计中最常用的运算，本章讨论有关查找的内容。本章首先介绍查找及查找表的有关概念、评价查找算法性能的指标等基本内容，然后重点讨论在顺序表、树表、散列表和索引表等几种数据表中查找特定元素的方法和算法，并对有关性能作必要的分析和比较。

什么是查找？简单地说，就是在数据集中找出一个“特定元素”。如日常生活中的查字典、电话号码、图书等，高考考生在考试后通过信息台查询成绩等。

在软件设计中，通常是将待查找的数据元素集以某种表的形式给出，从而构成一种新的数据结构——**查找表**。

在查找表中，每个元素可能由有多项信息组成，通常将每一项称为一个**字段**。例如，在学校的课程成绩表中，通常会包含如下几个字段：学号，姓名，成绩，备注等。在高考成绩表中，至少要包含如下几个字段：准考证号码、座位号、姓名、总分、四门单科成绩等。有初学者可能会问：有了姓名，还要学号或准考证号干什么？对这一问题，有经验的人自然会给出答复：“为了避免同名”。也正因为如此，通常在一个数据库中都会设置类似于学号或准考证号这样的能够标识元素的字段。

一般来说，在一个数据表中，若某字段(项、域)的值可以标识一个数据元素，则称之为**关键字（或键）**。也就是说，给定该关键字(项、域)的一个值，就可以标识（或对应到）一个数据元素。例如，在高考成绩表中，若给定某考生的准考证号码（一个字段），就可以对应到**一个**学生的成绩信息记录，然而，若给定该考生的姓名（一个字段），则可能因为有同

名而导致对应到多个考生的成绩记录。对此，给出以下的区分：若此关键字的每个值均可以唯一标识一个元素，则称之为**主关键字**。否则，若该关键字（的某个值）可以识别若干个记录(或元素)，则称之为**次关键字**。

这样，可以给出**查找**的定义：**对给定的一个关键字的值，在数据表中搜索出一个关键字的值等于该值的记录或元素**。若找到了指定的元素，则称为**查找成功**，通常是返回该元素在查找表中的位置，以便于能存取整个元素的信息。若表中不存在指定的元素，则称查找**不成功**，或称为**查找失败**，此时一般是返回一个能表示查找失败的值。

接下来的问题是：查找表以什么结构形式给出？如何在某种表中进行查找？关于查找表的形式，可以说有多种，本书主要介绍三类表——顺序表，树表和散列表，另外，还涉及到索引表结构。不同形式的表对应不同的查找方法，因而查找的时间性能也有所不同，这些是本章的重点部分。

查找算法的时间性能一般以查找长度来衡量。所谓**查找长度**是指查找一个元素所进行的关键字的比较次数。通常情况下，由于各元素的查找长度有所差异，因而常以**平均查找长度**、**最大查找长度**等来衡量查找算法的总的时间性能。

7.2 顺序表的查找

顺序表查找的**问题描述**：设查找表以一维数组来存储，要求在此表中查找出关键字的值为 x 的元素的位置，若查找成功，则返回其位置（即下标），否则，返回一个表示元素不存在的下标（如 0 或-1）。

在顺序表查找元素，根据元素之间是否具有递增（减）特性又可分为三种情况，即简单顺序查找、二分查找和分块有序查找。各种情况的查找方法及其性能存在较大差异，下面分别讨论。

7.2.1 简单顺序查找

简单顺序查找对数据的特性没有要求，即无论是否具有递增（减）特性均可以，因此，其查找只能从表的一端开始，逐个比较各元素，若成功，返回该记录(元素)的下标，否则返回 0，以表示失败。

说明：关于数据表元素下标的说明：虽然 C 语言中数组的下标是从 0 开始的，但考虑到简单顺序查找算法的特点，将数组中存储元素的下标范围约定为 1～n，因此，存储数组需要描述为 A[n+1]。这样，可以通过返回下标 0 来表示查找失败。

在实现查找时，搜索方向可以从下标 1 到 n，也可以从 n 到 1。为节省时间，采用后者。算法如下：

```
int seq_search(elementtype A[],int n, keytype x)
{
    i=n; A [0] .key=x;              //设定监视哨
    while (A [i] .key!=x )i--;
    return  i;
}
```

虽然元素的存储范围为 1～n，但该算法中还是利用了元素 A [0] ,这是一个小技巧，其

作用是充当**监视哨**：当查找失败时，肯定会在 A [0] 中“找到”该元素，因而返回其下标 0 以表示查找失败。若不设此监视哨，则在每次循环中均要判断下标（即 i 的值）是否越界。因而这样设置可以节省约一半的比较时间。

分析：该算法在查找某一元素时所作的比较次数取决于该元素在表中的位置，因而各元素的查找长度显然不同。为此，下面只讨论其平均查找长度。假设每个元素的查找概率相同，则由于各元素的查找长度依次取值从 1 到 n，因此，查找成功时的平均查找长度为：

ASL=(1+2+…+n)/n=(n+1)/2

很显然，**失败的查找长度**为 n+1。

对这样的查找长度，在 n 取值较小时还可以接受，但若表的规模较大，则难以接受。例如，在一本有 10 万个词汇的英语词典中查一个新单词时，若采用这种方法，平均需要比较 5 万个单词，这显然难以忍受。即使用计算机来查找，其时间耗费也是较大的。为此，需要讨论更快的查找方法。

7.2.2 有序表的二分查找

显然，我们在查英语词典类的数据表时，不会采用上述简单顺序查找方法，这是因为词典中的元素已经按英语字母的次序排列好了。更一般地说，如果查找表 A 已经按关键字递增(减)有序，此处不妨设为递增有序，则可采用**二分查找**（也叫**折半查找**）来查找。

1．二分查找方法

二分查找的**查找过程**如下：设查找区域的首尾下标分别用变量 low 和 high 表示（初值分别为 0 和 n-1），将待查关键字 x 和该区域的中间元素（其下标 mid 的值为 low 和 high 的算术平均值，即(low+high) / 2）的关键字进行比较，根据比较的结果分别作如下处理：

(1) x==A [mid] .key: 查找成功，返回 mid 的值。

(2) x<A [mid] .key: 说明元素只可能在左边区域(下标从 low 到 mid-1)，因此应在此区域继续查找。

(3) x>A [mid] .key: 说明元素只可能在右边区域(下标从 mid+1 到 high)，因此应在此区域继续查找。

若表中存在所要查找的元素，则经过反复执行上述过程可以很快地查找到，并返回元素下标。图 7-1 所示为在一个有序表中查找 8 的二分查找过程示意图。(其中数组的下标从 0 开始，这与简单顺序查找中略为有所不同。)

如果表中不存在所要查找的元素，如何能判断出来？在这种情况下，由于在查找过程中不断缩小查找区域（low 增大或 high 缩小），从而导致查找区域为空，即 low>high，即如果 low>high 成立，则表明查找区域为空，因而查找失败。由此得流程图如图 7-2 所示。

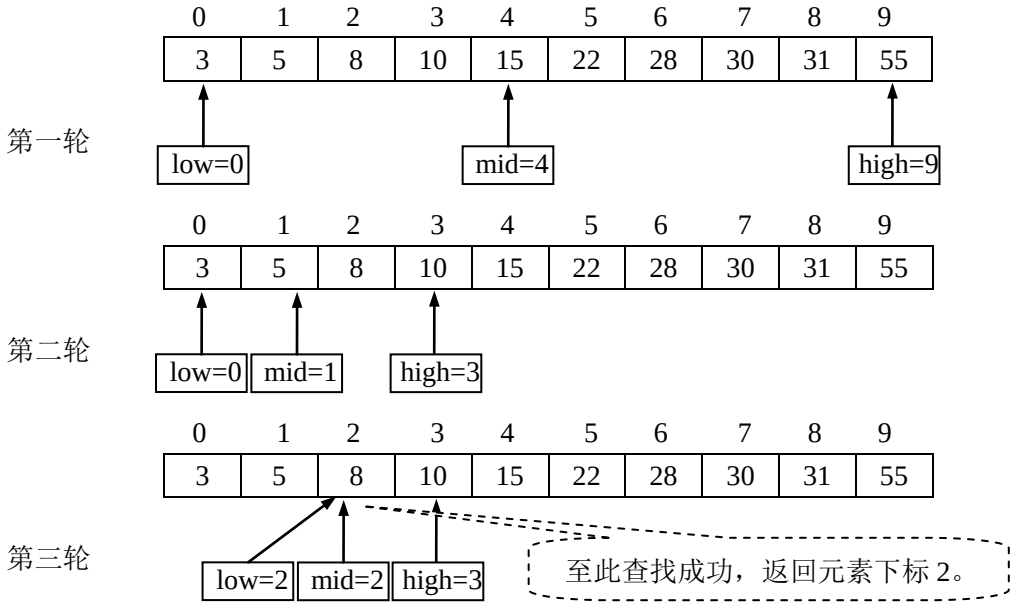


图 7-1 一个有序表中查找元素 8 的二分查找过程

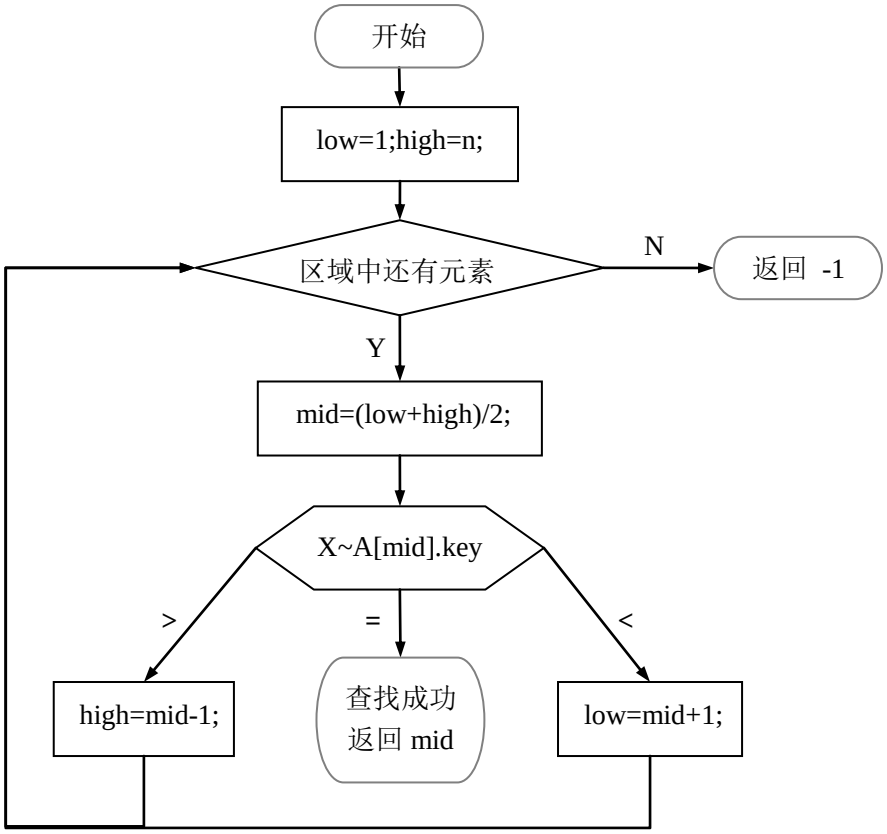


图 7-2 二分查找流程图

【二分查找算法描述】

```
int bin_search(elementtype A [ ] ,int n, keytype x)
{ int mid,low=0,high=n-1;        //初始化查找区域
  while ( low<=high)
    { mid=(low+high)/ 2;
      if (x==A [mid] .key) return  mid;
         else if (x<A [mid] .key) high=mid-1;
         else low=mid+1;
     }
    return  -1;     //返回查找失败的标志
}
```

由求解方法的描述可知，若当前中间元素不是所查找的元素时，其求解区域变成了原来区域的两个子区域中的一个，因而有二分查找（或折半查找）之名。

2．二分查找的递归算法

由于在子区域中的查找方法和原区域的查找方法相同，因此，二分查找算法也可采用递归方式来描述，而且这也是较常见的形式。下面讨论二分查找的递归算法。

由二分查找方法的描述可知，在给定的区域中查找元素时，根据与中间元素比较的结果分三种情况来处理，其中在相等时就直接返回地址，而在大于或小于中间元素时需要继续在子区域中查找。在子区域中查找时，除了查找范围不同外，查找方法与原区域的查找方法相同。

由此可知，在递归调用时，需要将查找区域即 low 和 high 作为算法的参数。

另外的一个问题是，这样的调用要进行到什么时候为止？如何判断查找失败？显然，算法结束于两种情况：

其一是查找成功－－在与某个中间元素相等时，返回该中间元素的下标并结束。

其二是查找失败，从而导致查找区域不断缩小，以至于为空，即 low>high。在这种情况下，显然要返回-1 以表示查找失败。由此可得算法如下：

```
int bin_search(elementtype A [ ] ,int low, int high, keytype x)
{  int mid;
  if ( low>high) return -1;              //查找失败
    else { mid=(low+high) / 2;           //求解中间元素的下标
          if (x==A [mid] .key) return  mid;   //查找成功
          else if (x<A [mid] .key)
                     return (bin_search(A,low,mid-1,x);  //在左边区域查找
                else  return (bin_search(A,mid+1,high,x); //在右边区域查找
         }
 }
```

3．算法分析

为便于描述二分查找算法的执行过程，并为了分析二分查找的算法性能，可以采用**二分**

查找的判定树——**一种**二叉树的形式来描述其查找过程。其构造过程如下：

对当前的查找区域 low～high，将其中间元素的下标 mid 作为根结点的值，将左边区域（即 low～mid-1）的查找过程所对应的二叉树作为其左子树，将右边区域（即 mid+1～high）的查找过程所对应的二叉树作为其右子树。

例如，对有 13 个元素的有序表的二分查找的判定树如图 7-3 所示，其中每个结点上标出了查找区域的首尾下标。由该树可以看出，在其中查找 A［10］依次比较的元素为 A［6］，A［9］，A［11］，A［10］，如图上双线表示的路径。

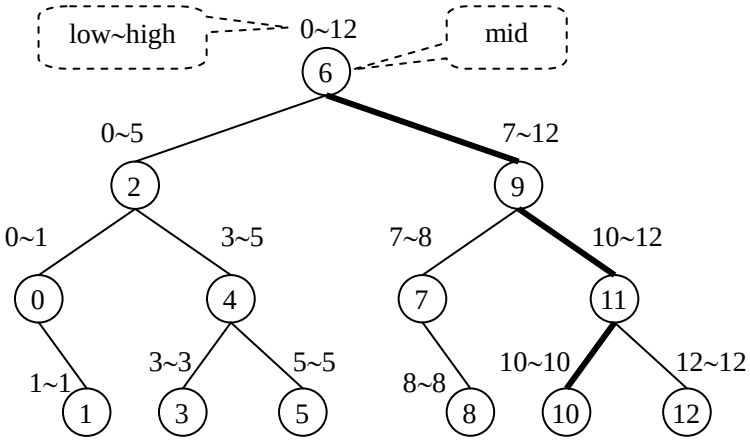


图 7-3 有 13 个结点的二分查找判定树

在数据表中查找任一元素的查找长度取决于该结点在相应的判定树上的层次数，因而不会超过该树的深度。由于有 n 个结点的判定树和有 n 个结点的完全二叉树具有相同的高度，因此，任一元素的查找长度不超过$\lfloor \log_2^n \rfloor + 1$，因而其平均查找长度也小于该值。事实上，在等概率情况下，二分查找算法的平均查找长度为 $\log_2^{(n+1)}-1$。（计算过程略）

7.2.3 索引顺序表的查找

在许多情况下，可能会遇到这样的表：整个表中的元素未必（递增/递减）有序，但若划分为若干块后，每一块中的所有元素均小于(或大于)其后面块中的所有元素。称这种特性为**分块有序**。

对分块有序表的查找，显然不能采用二分查找方法，但如果采用简单顺序查找方法查找，又太浪费时间，因为没有充分利用所给出的条件。在这种情况下，可为该顺序表建立一个**索引表**。索引表中为每一块设置一**索引项**，每一索引项包括两部分：该块的起始地址和该块中最大(或最小)关键字的值（或者是所限定的最大（小）关键字）。将这些索引项按顺序排列成一有序表，即为索引表。

例如，在图 7-4 所示的结构中，数据表可划分为 5 块，每个索引项中存放对应块中最大或限定的最大值。第一块中的最大值为 10，第四块索引项中的最大值为 40，但此值不存在，所以可以认为是限定的最大值，而不必一定要在表中出现。

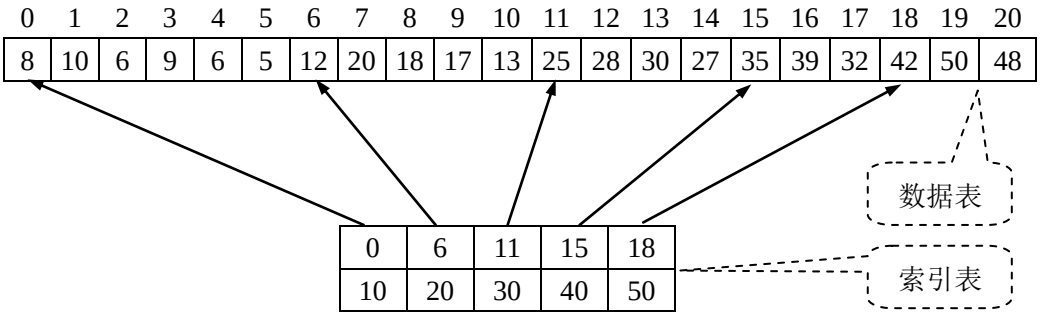


图 7-4 索引表结构示例

在这种结构中的查找要分两步进行：首先要通过在索引表中查找以确定元素所在的块，然后在所确定的块中进行查找。由于索引表是按关键字递增（或递减）有序的，因此，在索引表中的查找既可以采用简单顺序方式的查找，也可以采用二分查找，这要取决于索引表的项数：如果项数较多，则采用二分查找是合适的，否则，采用顺序查找就可以了。在块内的查找由于块内元素的无序而只能采用简单顺序查找。

例如，在图 7-4 所示的表中，如果要查找 35 这个元素，由索引表可知该元素应在第四块中，因而其查找区域为 15～17（由第四块及第五块的两个索引项中的首地址所确定），然后在这一区域中按简单顺序方式来查找，确定其地址为 15。但如果要搜索元素 37，也在这一区域中搜索，但搜索失败。算法从略。

分析：这种查找的时间性能取决于两步查找时间之和：如前所述，第一步可用简单顺序方式和二分查找方法之一进行，第二步只能采用顺序查找，但由于子表长度较小，因此其时间性能介于顺序查找和二分查找之间。

7.3 树表的查找

如前所述，在递增(减)有序表中采用二分查找算法查找的速度是比较快的，但也存在问题：若要在其中插入或删除元素时，需要移动元素以保持其有序性。若这种插入和删除是经常性的运算，则较浪费时间。为此，可采用动态链表结构。二叉排序树便是一种合适的结构，下面介绍这种结构及其上的查找运算以及相关的操作。

7.3.1 二叉排序树

1. 二叉排序树的定义及其查找

定义：**二叉排序树**是一棵二叉树，或者为空，或者满足如下条件：

① 若左子树不空，则左子树上所有结点的值均小于根的值。

② 若右子树不空，则右子树上所有结点的值均大于根的值。

③ 其左右子树均为二叉排序树。

图 7-5 所示二叉树即为一棵二叉排序树。

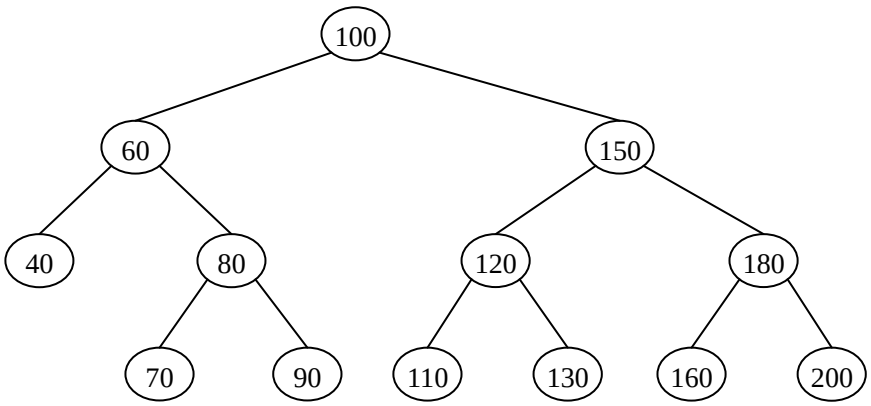


图 7-5　二叉排序树示例

由定义可知，二叉排序树中以任一结点为根的子树均为二叉排序树。由此可知二叉排序树的一个**特性**：二叉排序树的中序序列是递增序列。

在二叉排序树中**查找**值为 x 的结点，可依据其与根结点(*T)的值的关系分别处理：（为简便起见，假设每个结点中的值就是关键字）

① x==T->key：查找成功，返回指针 T 即可。

② x<T->data：元素只可能在左子树中，因而需在左子树中继续查找。

③ x>T->data：元素只可能在右子树中，因而需在右子树中继续查找。

若树中存在待查元素，则按照这种方式反复搜索一定能找到。

例如，在图 7-5 所示二叉排序树中查找 110 时，依次比较 100，150,120 和 110 即可完成查找。反之，若不存在待查元素，则将搜索到空指针。例如，要在图 7-5 中搜索元素 65，就会在依次比较 100，60，80，70 之后搜索到 70 所在结点的左边，从而使搜索指针为空，故查找失败。

由查找过程的分析可知，在二叉排序树中查找特定元素的算法既可采用递归形式，也可采用非递归形式。下面分别给出该查找的递归形式和非递归形式的算法。

(1)非递归算法

在非递归形式的算法中，需要设置一个指针（不妨设为 P）依次指示所比较的元素，显然，其初值为根指针 T。算法如下：

```
Bnode *bst_search(Bnode *T; keytype x)
{
  Bnode *P=T;          //P 指向根
  while (P!=NULL)
      if(x==P->key) return  P;             //查找成功
         else if (x<P->key) P=P->lchild;   //到左子树中继续查找
                      else  P=P->rchild;   //到右子树中继续查找
   return  P;            //返回结果，既可能为空，也可能非空
  }
```

(2)递归算法

在递归算法中，在左右子树中的查找是通过递归调用来实现的：

```
Bnode *bst_search(Bnode *T; keytype x)
{
  if (T==NULL || t->key==x) return  T;  //子树为空，或者已经找到时均可结束
  else if (x<T->key)
          return(bst_search(T->lchild,x));
                    //将在左子树中查找的结果作为函数的结果返回
      else return(bst_search(T->rchild,x));
                    //将在右子树中查找的结果作为函数的结果返回
}
```

显然，二叉排序树中某结点的查找长度等于该结点的层次数。

2. 二叉排序树的构造和维护

由于二叉排序树是动态结构，因而其构造是通过逐个插入结点来实现的。为此，下面先讨论在二叉排序树中插入结点的过程及其算法，在此基础上讨论二叉排序树的建立。

在往二叉排序树中插入结点时，为保持其二叉排序树的特征，须根据其值的具体情况分别处理：

① 若该值小于根结点的值，则应往左子树中插入，因而可通过调用相同的算法(即递归调用插入算法)来实现往左子树中的插入。

② 若该值大于根结点的值，则可通过递归调用插入算法实现往右子树中的插入。

按照这样的方式递归调用若干次后，总可以搜索到一个空子树位置，这就是要插入的位置，可将该结点放在该子树上，作为该子树的根结点并连到其父结点上。

例如，要在图 7-5 所示的二叉排序树中插入值为 75 的结点，所执行的操作过程如下：

－－首先和根结点（即值为 100 的结点）比较，由于 75 比 100 小，故要递归调用插入算法往其左子树（根为 60）中插入；

－－由于 75 比 60 大，故要递归调用插入算法往其右子树（根为 80）中插入；

－－由于 75 比 80 小，故要递归调用插入算法往其左子树（根为 70）中插入；

－－由于 75 比 70 大，故要递归调用插入算法往其右子树中插入。由于其右子树为空，故可将该结点直接插入到 70 的右边，作为其右孩子，插入完毕。

依据上述描述，可写出插入结点的递归算法如下：

```
void insert(Bnode **T, Bnode *S) //将指针 S 所指结点插入到二叉排序树 T 中
  {
    if (*T==NULL )*T=S;        //插入到空树时，插入结点成为根结点
    else if (S->key<(*T)->key)
              insert(&((*T)->lchild),S); //插入到 T 的左子树中
         else insert(&((*T)->rchild),S); //插入到 T 的右子树中
   }
```

二叉排序树的构造是通过从空树出发，依次插入结点来实现的。在一系列插入结点操作之后，可生成一棵二叉排序树(算法可参照建单链表的算法)。例如，若输入序列为 100，80，85，70，130，150，120，90，50，110，160，则可生成二叉排序树如图 7－6 所示：

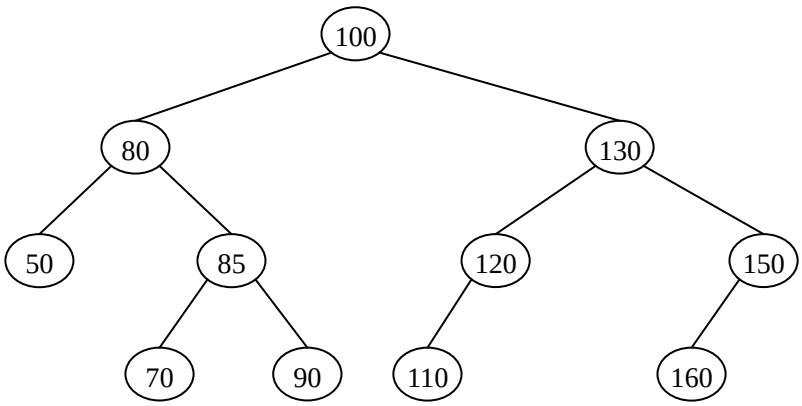


图 7-6　建立二叉排序树示例

7.3.2 平衡二叉树

为了使二叉排序树的平均查找长度更小，需要让各结点的深度尽可能小，因此，树中每个结点的两个子树的高度不要偏差太大，由此而出现了平衡二叉树。

1.平衡二叉树的概念

平衡二叉树（balance binary tree）又称 AVL 树，定义如下：

平衡二叉树是一颗二叉树，或者为空，或者满足如下性质：

① 左、右子树高度之差的绝对值不超过 1；

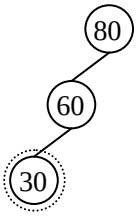
② 左、右子树都是平衡二叉树。

由定义可知，如果一棵二叉树是平衡二叉树，则其中每个结点的左、右子树的高度之差的绝对值不超过 1，从而使得各结点的平均查找长度较小，保证了良好的检索效率。下面讨论平衡二叉树的构造和调整。为了便于问题描述，给出结点的**平衡因子**的概念：

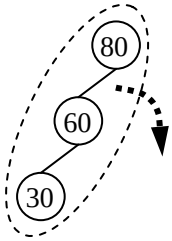
结点的平衡因子 = 结点的左子树高度 - 结点的右子树高度

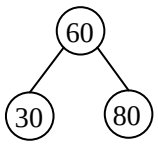
2.平衡化

如何构造平衡二叉树？G.M.Adelson 和 E.M.Landis 在 1962 年提出了的一种经典调整方法。下面先以实例对有关调整操作产生一个感性认识，然后再介绍这种调整方法。整个调整过程是通过在一棵平衡的二叉排序树中（按照二叉排序树的方式）依次插入元素，若出现不平衡，则根据二叉排序树的特性以及插入的位置作适当调整。假定插入序列为：80、60、30、45、50、85、90。

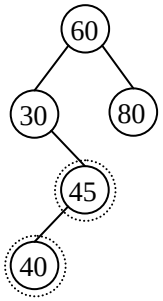


不平衡，对此，应以 60 为根结点，其余结点按照大小关系调整，使之平衡。

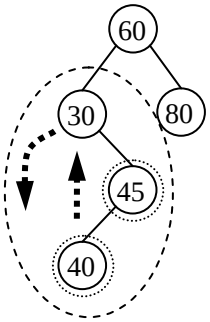


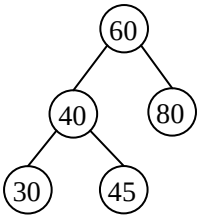


（a）依次插入 80、60、30 后　　调整过程示意图　　调整后

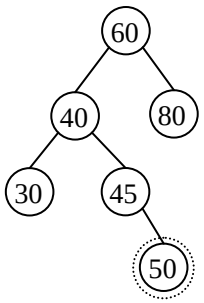


此时，有 2 各结点不平衡（60 和 30），只要将以 30 为根结点的子树调整好即可，即将 40 调整为子树根结点，其余结点按大小关系调整。

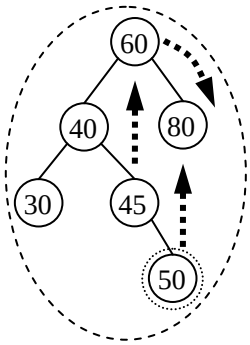


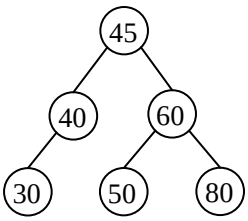


（b）依次插入 45 和 40 后　　调整过程示意图　　调整后

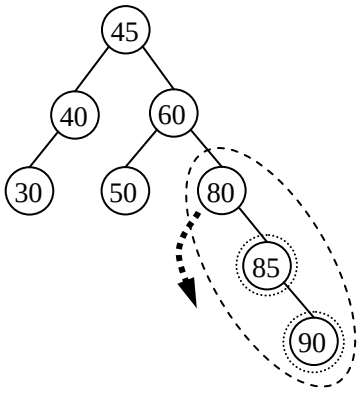


此时，结点 60 不平衡，太偏向一边，因而需要选择的元素的大小最好处于靠近中间的，直觉上，40 和 45 合适，不妨选择 45 作为新的根结点，其它结点按大小关系调整。

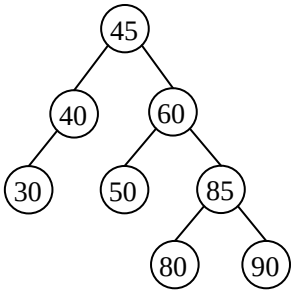




（c）插入 50 后　　调整过程示意图　　调整后



此时，有 3 各结点不平衡（45、60 和 80），只要将以 80 为根结点的子树调整好即可，即将 85 调整为子树根结点，其余结点按大小关系调整。



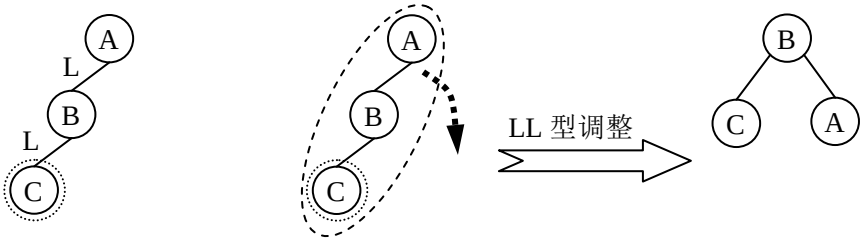
（d）依次插入 85 和 90 后　　调整后

图 7-7 平衡二叉树调整过程示例

如前所述，G.M.Adelson 和 E.M.Landis 提出的调整方法的基本思想是逐个地在平衡的二叉树上插入结点，当出现不平衡时及时调整，以保持平衡二叉树的性质。调整操作的方法根据新插入结点与最低不平衡结点（不妨用 A 表示，即 A 的祖先结点可能有不平衡的，但其所有后代结点都是平衡的）的位置关系分为 LL 型、RR 型、LR 型和 RL 型 4 种类型分别处理。各种调整方法如下：

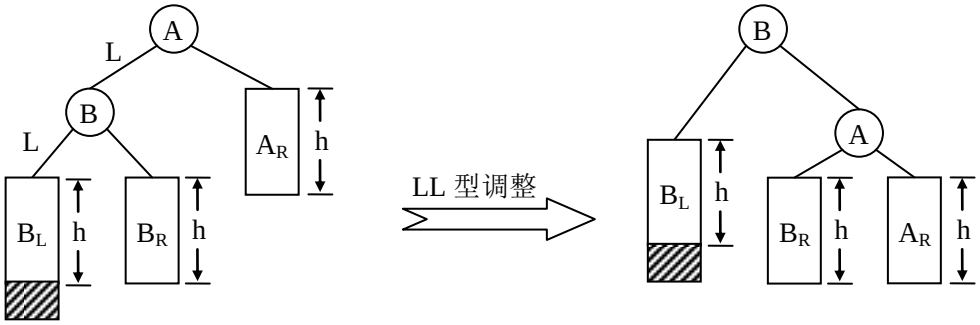
（1）LL 型调整

由于在 A 的左孩子（L）的左子树（L）上插入新结点，使原来平衡的二叉树变得不平衡，并且使 A 成为最低的不平衡结点，即 A 的平衡因子由 1 变为 2，使得以 A 为根的子树失去平衡，如图 7-8 和图 7-9 所示。



（a）插入结点 C 导致不平衡　（b）LL 型调整示意图　（c）LL 型调整结果

图 7-8 最简单的 LL 型调整示意图



（a）调整前的结构形式　（b）调整后的结构形式

图 7-9 LL 型调整的一般形式示意图

其中，图 7-8 所示为这种类型调整的最简单形式，表示 A 的左孩子的 B 的左子树（插入前为空）中插入结点 C 而导致不平衡。显然，按照大小关系，结点 B 应作为新的根结点，其余两个结点分别作为其左右孩子结点才能平衡，而 A 结点好像是绕结点 B 顺时针旋转了一样。

LL 型调整操作的更为一般的形式如图 7-9 所示，表示在结点 A 的左孩子 B 的左子树 B_L（不一定为空）中插入新结点（阴影部分表示）而导致不平衡。其中，各个子树的高度如图 7-9 中所示。

针对这种情况的调整方法包括如下 3 部分：

① 将 A 的左孩子 B 提升为新的根结点；

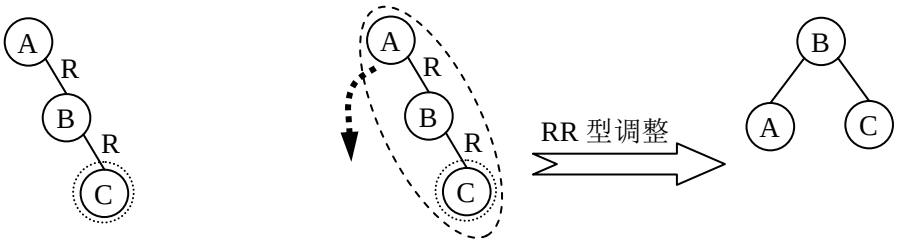
② 将原来的根结点 A 降为新的根结点 B 的右孩子；

③ 各子树按照大小关系重新连接，其中 B_L 和 A_R 连接关系不变，而 B_R 调整为 A 的左子树。

检查一下调整前后各个结点和子树的大小关系（中序序列）可知，调整操作不仅调整了各子树的高度以保持平衡，而且还保持了二叉排序树的的特性，即中序序列的不变性。

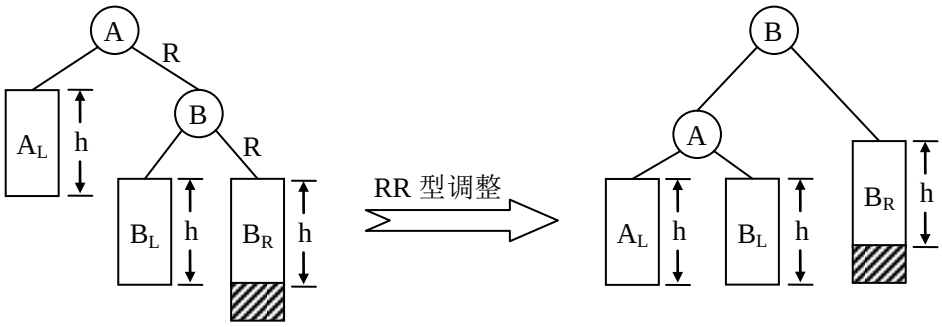
（2）RR 型调整

由于在 A 的右孩子（R）的右子树（R）上插入新结点，使 A 的平衡因子由-1 变为-2，致使以 A 为根的子树失去平衡，如图 7-10 和图 7-11 所示。



（a）插入结点 C 导致不平衡　（b）RR 型调整示意图　（c）RR 型调整结果

图 7-10 最简单的 RR 型调整示意图



（a）调整前的结构形式　（b）调整后的结构形式

图 7-11 RR 型调整的一般形式示意图

该类调整与 LL 型调整对称，其中，图 7-10 所示为这种类型调整的最简单形式，表示在 A 的右孩子 B 的右子树（插入前为空）中插入新结点 C 而导致不平衡。显然，按照大小关系，结点 B 应作为新的根结点，其余 2 个结点分别作为其左右孩子结点才能平衡，而 A 结点好像是绕结点 B 逆时针旋转一样。

RR 型调整操作更为一般的形式如图 7-11 所示，表示 A 的右孩子 B 的右子树 B_R（不一定为空）中插入新结点（阴影部分表示）而导致不平衡。其中，各个子树的高度如图 7-11 中所示。针对这种情况的调整方法包括如下 3 个部分:

① 将 A 的右孩子 B 提升为新的根结点；

② 将原来的根结点 A 降为新的根结点 B 的左孩子；

③ 各子树按大小关系重新连接，其中，A_L 和 B_R 连接关系不变，而 B_L 调整为 A 的右子树。

（3）LR 型调整

由于在 A 的左孩子（L）的右子树（R）中插入新结点，使 A 的平衡因子由 1 变为 2，致使以 A 为根的子树失去平衡，如图 7-12 和图 7-13 所示。

其中，图 7-12 所示为这种类型的最简单形式，表示在 A 的左孩子 B 的右子树（插入前为空）中插入新结点 C 而导致不平衡。显然，按照大小关系，C 应作为新的根结点，其余 2 个结点分别作为 C 的左右孩子结点才能平衡。

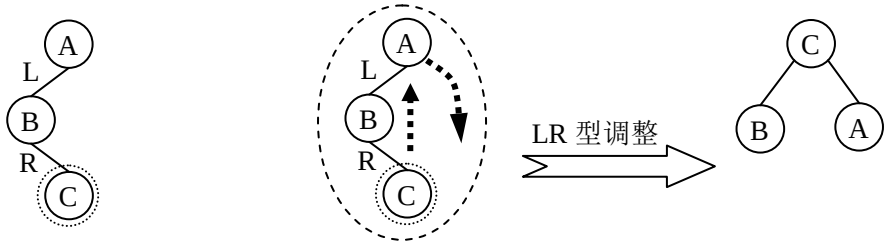
LR 型调整操作的更为一般的形式如图 7-13 所示，表示在 A 的左孩子 B 的右子树（根结点为 C，不一定为空）中插入新结点（两个阴影部分之一）而导致不平衡。其中，各个子树

的高度如图 7-13 中所示。针对这种情况的调整方法包括如下 3 个部分：

① 将 C 提升为新的根结点；

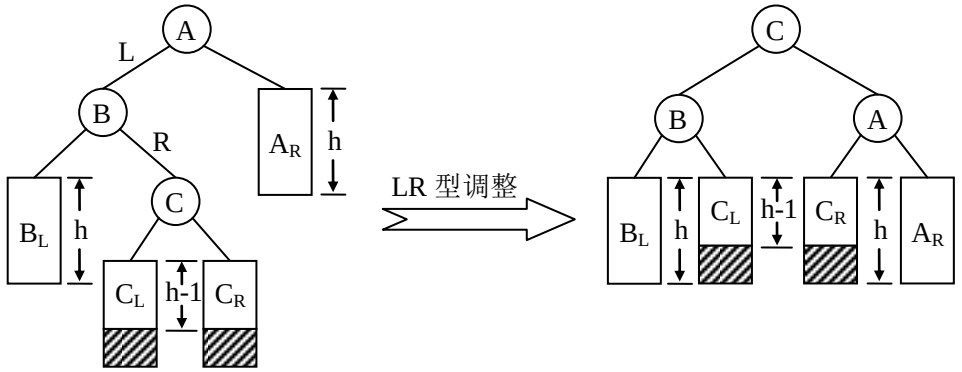
② 将原来的根结点 A 降为新的根结点 C 的右孩子；

③ 各子树按大小关系重新连接，其中，B_L 和 A_R 连接关系不变，而 C_L 调整为 B 的右子树，C_R 调整为 A 的左子树。



（a）插入结点 C 导致不平衡　（b）LR 型调整示意图　（c）LR 型调整结果

图 7-12 最简单的 LR 型调整示意图

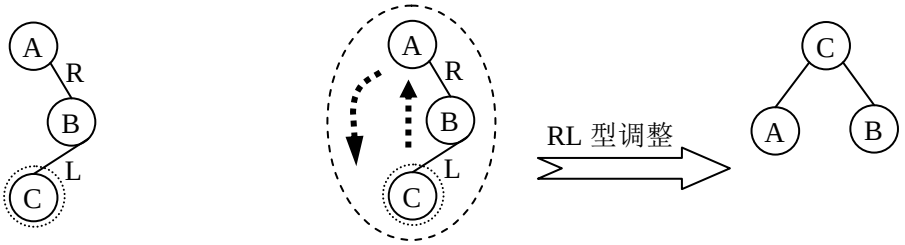


（a）调整前的结构形式　（b）调整后的结构形式

图 7-13 LR 型调整的一般形式示意图

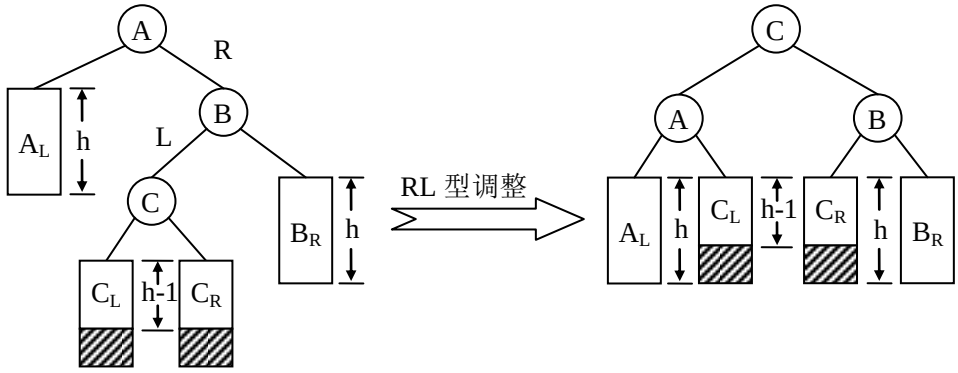
（4）RL 型调整

由于在 A 的右孩子（R）的左子树（L）中插入新结点，使 A 的平衡因子由-1 变为-2，致使以 A 为根的子树失去平衡，如图 7-14 和图 7-15 所示。



（a）插入结点 C 导致不平衡　（b）RL 型调整示意图　（c）RL 型调整结果

图 7-14 最简单的 RL 型调整示意图



（a）调整前的结构形式　　（b）调整后的结构形式

图 7-15 RL 型调整的一般形式示意图

这类调整与 LR 对称，其中，图 7-14 所示为这种类型的最简单形式，表示在 A 的右孩子 B 的左子树（插入前为空）中插入新结点 C 而导致不平衡。显然，按照大小关系，C 应作为新的根结点，其余 2 个结点分别作为 C 的左右孩子结点才能平衡。

RL 型调整操作的更为一般的形式如图 7-15 所示，表示在 A 的右孩子 B 的左子树（根结点为 C，不一定为空）中插入新结点（两个阴影部分之一）而导致不平衡。其中，各个子树的高度如图 7-15 中所示。针对这种情况的调整方法包括如下 3 个部分：

① 将 C 提升为新的根结点；

② 将原来的根结点 A 降为新的根结点 C 的左孩子；

③ 各子树按大小关系重新连接，其中，A_L 和 B_R 连接关系不变，而 C_L 调整为 A 的右子树，C_R 调整为 B 的左子树。

平衡的二叉树排序树的高度接近 $\log_2 n$ 的数量级，从而保证了在二叉排序树上插入、删除和查找等基本操作的平均时间复杂度为 $O(\log_2 n)$。

7.4　散列表的查找

7.4.1 哈希表的基本概念

前面两类表的查找均是基于比较运算来实现的，即通过比较元素的值来确定下一次查找的位置。然而，在现实中，有很多查找是直接通过计算来实现的，即对给定的关键字 key，用一个函数 H(key)来计算出该关键字所标识元素的地址。例如，在如表 7-1 所示的成绩表中，若以学号为关键字，则对给定的关键字 20120503130，可通过将其末两位 30 转换为元素在表中的地址 30 来实现。

表 7-1 成绩表示例

序号	学号	姓　名	成　绩	备　　注
1	20120503101	王　云	80	
2	20120503102	李　明	92	
…	…	…	…	
30	20120503130	张　敬	83	

…	…	…	…	
40	20120503140	李承业	92	

在用函数 H 计算给定关键字 key 的地址时，称函数 H 为**哈希(Hash)函数**，或**散列函数**，称计算出的地址 H(key)为**散列地址**，按这种方法建立的表称为**哈希表**或**散列表**。理想情况下，散列函数在关键字和地址之间建立一一对应关系，从而使得查找只须一次计算即可完成。

然而，在许多情况下，并非这么理想：由于关键字值的某种随机性，使得这种一一对应关系难以发现或构造。因而可能会出现不同的关键字对应一个存储地址的情况，即 k1≠k2，但 H(k1)=H(k2)。称这种现象为**冲突**，此时的 k1 和 k2 为**同义词**。例如，对大家所熟知的汉字的输入编码，若用拼音方式来输入汉字，则许多汉字会对应相同的编码(即同音字)。

显然，冲突影响哈希表的构造及查找。为此，要选择一个恰当的哈希函数，以避免冲突。然而，在大多数情况下，冲突是不可能完全避免的，这是因为所有可能的关键字的集合可能比较大，而对应的地址数则可能比较少。为此，需从两个方面着手：一方面，选择好的散列函数以使冲突尽可能少地发生。另一方面，须妥善处理出现的冲突。下面分别简要讨论有关问题。

7.4.2 哈希函数的构造方法

构造哈希函数的方法很多。作为一个好的散列函数，应能使冲突尽可能地少，因而应具有较好的随机性，这样可使一组关键字的散列地址均匀地分布在整个地址空间。由于构造散列表具有较大的主观性，并且需要有一定的经验，故初学者在开始学习时可能会感到较抽象，故本小节内容能简要了解即可。

常用的构造散列函数的方法有：

① 直接定址法

取关键字的某个线性函数值作为散列地址，即 H(k)=a*k+b (a,b 为常数)。这种方法是一种较为直观的方法。

② 除留余数法

取关键字被某个不大于表长 m 的数 P 除后所得的余数作为散列地址。即 H(k)=k % P(P≤m)

这是一种较简单，也是较常见的构造方法。一般来说，在 P 取值为素数（质数）时，冲突的可能性相对较少。

③ 平方取中法

取关键字平方后的中间几位作为散列地址（若超出范围时，可再取模）。这种方法使得关键字的每一位的取值都会影响到地址，从而减少冲突。

④ 折叠法

这是在关键字的位数较多（如身份证号码），而地址区间较小时，常采用的一种方法。

这种方法是将关键字分隔成位数相同的几部分(最后一部分不够时，可以补 0)，然后将这几部分的叠加和作为散列地址(若超出范围，可取模)。具体叠加方法可以有多种，如每段最后一位对齐，或相邻两段首尾对齐等。

⑤ 数值分析法

如果事先知道所有可能的关键字的取值时，可通过对这些关键字分析，发现其变化规律，构造出相应的函数。

7.4.3 处理冲突的方法

如上所述，冲突不可能完全避免，因此，妥善处理冲突是构造散列表必须要解决的问题。

假设散列表的地址范围为 0～m-1，当对给定的关键字 k，由散列函数 H(k)计算出的位置上已有元素时，则出现冲突。此时必须为该元素另外找一个空的位置。如何确定这一空的位置？对此，有几种最常用的处理方法：

1. **开放定址法**

当由 H(k)算出的位置不空时，依次用下面函数以找出一个新的空位置。

$H_i(k)=(H(k)+d_i)$ mod m，其中 i=1,2,…，k(k≤m-1)

其中 d_i 的取值常用以下两种形式之一，从而得到两种典型的处理方法：

(1) **线性探测法**

$d_i=i$,即 d_i 依次取 1，2，…，因此 $H_i(k)=(H(k)+i)$ % m。

换句话说，在这种情况下，从 H(k)开始往后逐个搜索空位置，若后面没有空位置，就从头开始搜索(直到搜索到空位，或者回到 H(k)为止)。故称这种搜索方法为**线性探测法**。

例：已知散列表的地址区间为 0～11，散列函数为 H(k)=k % 11,采用线性探测法处理冲突，试将关键字序列 20,30,70,15,8,12,18,63,19 依次存储到散列表中，构造出该散列表，并求出在等概论情况下的平均查找长度。

解：为构造散列表，需要计算每个元素的散列地址，并根据所计算出的位置中是否已经有元素而作不同的处理：如果还没有存放元素，则将元素直接存放进去，否则，就往后搜索空位置，如果后面没有空位置，就绕到最前面重新搜索，直到找到空位置，再将该元素存放进去即可。为便于描述，假设数组为 A。

本题中各元素的存放过程如下：

H(20)=9, 可直接存放到 A[9]中去。

H(30)=8， 可直接存放到 A[8]中去。

H(70)=4， 可直接存放到 A[4]中去。

H(15)=4， 因为 A[4]已经被 70 占用，故往后搜索到 A[5]并存放。

H(8)=8， 因为 A[8]、A[9]已分别被 30、20 占用，故往后搜索到 A[10]并存放。

H(12)=1, 可直接存放到 A[1]中去。

H(18)=7, 可直接存放到 A[7]中去。

H(63)=8， 因为 A[8]、A[9]、A[10]均被占用，故往后搜索到 A[11]并存放。

H(19)=8， 因为下标为 8~11 的元素均已被占用，故往后搜索并绕回到 A[0]存放。

另外，为便于计算所要求的平均查找长度，需要知道每个元素的查找长度。为此，在放置每个元素到散列表的同时，将其搜索次数标注在元素的下方，这同时也是该元素的查找长度。例如，元素 12 是直接存放在 A[1]中的，故其查找长度为 1，因而在其下面标 1。而 63 这个元素是从 A[8]开始逐个搜索到 A[11]的，也就是说，其比较次数是 4，因而在其下面标

4。由此得散列表如下：

0	1	2	3	4	5	6	7	8	9	10	11
19	12			70	15		18	30	20	8	63
5	1			1	2		1	1	1	3	4

搜索次数

图 7-16 散列表示例

平均查找长度的求解：由于各元素的查找长度已经标注在元素的下方，因此容易求出在等概率情况下，该表成功的平均查找长度如下：

（1×5＋2×1＋3×1＋4×1＋5×1）/9＝19/9

在运用线性探测法处理冲突时，可能会出现这样的情况：某一连续的存储区已经存放满了，因此在经过这一区域往后搜索空位置时，需要比较较多的元素，从而导致查找长度增大。下面的二次探测法可改善这一问题。

(2) **二次探测法**

$d_i=\pm 1^2, \pm 2^2, \pm 3^2, \cdots\cdots, \pm k^2 (k \leqslant m/2)$，因此 $H_i(k)=(H(k)+i) \% m$。

也就是说，该方法是在原定位置的两边交替地搜索，其偏移位置是次数的平方，故称这种方法为**二次探测法**。

2. **再散列法**

当出现冲突时，也可这样处理：用另外不同的散列函数来计算散列地址。若此时还有冲突，则再用另外的散列函数，依次类推，直至找到空位置。也就是要依次用 $H_1(k), H_2(k), \cdots\cdots, H_l(k), (l=1,2,\cdots)$来搜索空位置。

3. **链地址法(拉链法)**

也称**开散列表**。这一方法是将所有冲突的记录（同义词）存储在一个链表中，并将这些链表的表头指针放在数组中(下标从 0 到 m-1)。这类似于图结构中的邻接表存储结构和树结构的孩子链表结构。

例：设散列函数为 H(k)=k % 11，采用拉链法处理冲突，将上例中关键字序列依次存储到散列表中，并求出在等概论情况下的平均查找长度。

解：通过计算各元素的散列函数得到开散列表如图 7-17 所示。

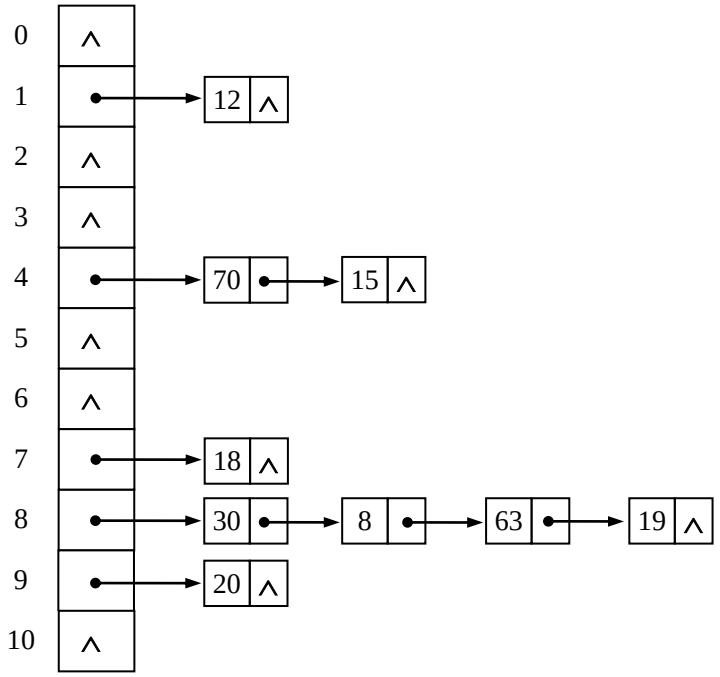


图 7-17 拉链法处理冲突示例

平均查找长度的求解：由于在各链表中的第一个元素的查找长度为 1，第二个元素的查找长度为 2，依此类推可知其余各元素的查找长度，因此，在等概率情况下成功的平均查找长度为

(1×5+2×2+3×1+4×1)/9=16/9

显然，这一长度小于前面例题中的长度，读者可分析其内在的原因。

7.4.4 散列表的查找

在散列表中查找元素的过程和构造的过程基本一致：对给定关键字 k，由散列函数 H 计算出该元素的地址 H(k)。若表中该位置为空，则查找失败。否则，比较关键字，若相等，查找成功，否则根据构造表时所采用的处理方法找下一个地址，直至找到关键字等于 k 的元素（成功）或者找到空位置（失败）为止。

例如，在第一个例中查找关键字为 19 的元素时，首先和 A[H(19)=8]中的元素比较，然后按线性探测法依次和数组 A 中下标为 9，10，11，0 的元素比较，直至成功。共进行了 5 次比较，和该元素下方所标的数字一致。

若要在其中查找一个不存在的关键字，如 21，则经过依次和数组 A 中下标为 10，11，0，1 的元素比较，在 A[2]中发现是空，从而判断查找失败。

由此可知在表中查找一个元素所进行的元素比较的次数和构成时的探测次数一样。因此，在第一例的表中查找一个元素，在等概率情况下的平均查找长度为（1×5＋2×1＋3×1＋4×1＋5×1）/9＝19/9。

类似地在第二例的表中查找一个元素，在等概率下的平均查找长度为 (1×5+2×2+3×1+4×1)/9=16/9

一般来说，在用链地址法构造的表中进行查找，比在用开放定址法构造的表中进行查找，其查找长度要小。

本章小结

查找是软件设计中最常用的运算，是在数据表中找出给定关键字为特定值的元素。查找长度是描述查找算法时间性能的指标。根据数据集的组织方式，有顺序表、树表、散列表和索引表等几种表，每种结构有其相应的查找方法。

对顺序表来说，根据元素间是否有递增（或递减）关系，有三种相应的查找方法：

如果没有给出元素间的大小关系，则只能采用简单顺序查找，这种方法的查找长度较大，等概率情况下，成功的平均查找长度是（n+1）/2，失败的查找长度是 n+1。

如果数据表中的元素具有递增（或递减）关系，则可采用二分查找方法查找，其时间性能较好，成功的平均查找长度是 $\log_2^{(n+1)}-1$，最大的查找是 $\log_2^{(n+1)}+1$。

如果数据表分块有序，也就是整个表不具有递增（或递减）关系，但在划分为若干块后，块内无序，块间有序。对这样的表可建一个索引表，相应的查找分两步走，其一是确定元素所在的块，其二是在所确定的块中顺序查找。查找的时间性能介于简单顺序查找和二分查找算法之间。

二叉排序树是特殊的二叉树，其中每个结点的值大于其左子树中所有结点的值，小于其右子树中所有结点的值。在二叉排序树中查找元素就是依据其大小关系进行的。构造二叉排序树是通过逐个插入元素来进行的。

散列表是通过散列函数计算出元素在表中的地址的。由于有冲突出现，故需要选择好的散列函数以减少冲突，另一方面，需要妥善处理冲突。线性探测法和开散列法是两种常用的处理冲突的方法。